

## Block 2, Part 3: Client side: Validation and feedback

---

*Peter Thomson (2017), Dave McIntyre (2019) and Stephen Rice (2021, 2022)*

### 1 Introduction

So far you have started working with JavaScript, examined the Document Object Model (DOM) and HTML forms, and we have introduced some JavaScript processing of form fields as well as the use of events to initiate processing.

In this part, we will focus on using forms as the main user interface for entering data into a web application.

There are two key aspects to this: the use of the correct element and attributes to restrict what the user can enter, and then the use of JavaScript to check that form data is generally correct on the client side offering useful feedback prior to submission to the server.

**Note that the code and the comments in the code are very important parts of the text of this document.**

#### For this part of Block 2

Before you start work on the practical activities of this part, you will need to download the files from Block 2 Part 3 resources; the files are contained in a zip file, so extract these and save them to your work folder.

### 2 Learning outcomes

On successfully completing this part's study, you should be able to:

- select appropriate form input elements for user interaction
- discuss the benefits of validating form data on the client side
- outline different approaches to providing feedback to a user who is completing a form, and to produce an appropriate HTML or JavaScript solution
- utilise and adapt a set of JavaScript or HTML features for validating different types of form inputs
- validate data entered into a form using a range of approaches, including the matching of regular expressions.

## 3 Form input elements

You were introduced to HTML5 form input elements in Block 1 Part 6. In this part, we will further experiment in practical terms with form elements and their functionality.

Note, while it is possible to create the user interface by just using text input elements for each input, that approach would allow the user to enter anything they like. It is better to use an input element that restricts the user to the type of value that you want. The type of input element also guides the user in entering their data and the form becomes much more intuitive to use.

In the following sections, we'll experiment with some HTML input types and we'll see how JavaScript can be used for client-side data validation.

### 3.1 Types of form input element

#### Activity 1 Checking your browser

 Allow 15 minutes for this activity

In your browser, view the **html-forms.html** file that you have downloaded from Block 2 Part 3 resources.

This file includes some of the inputs that you encountered in Block 1. Each of these is on a HTML form which submits data to a 'reflector' page – this page presents the data it is sent to help you understand what the form is doing.

Try the various input types and observe the values they permit you to enter and the values they submit to the web server as shown in the response from the reflector.

The file contains many examples, so we have also provided 'snippets' of each of the forms we discuss. We recommend you use those to explore the individual elements so you can focus on one element at a time. You will find those individual parts contained within the **form-snippets.zip** file. Unzip this file to a new folder. The snippets contain just the form and input elements and will not validate as HTML pages, but will function perfectly well for your needs.

**Whenever you encounter HTML code or JavaScript code, it is worth experimenting by changing parameters to help understand their effect.** We'll do this in the following activities.

#### The text input element

You may examine the code in **html-forms.html**, but this part is contained in the snippet **text\_input.html** and we recommend using that. It contains the following code:


```
<form method="POST" action="https://students.open.ac.uk/mct/tt284/reflect/reflect.php">
  <p>
    <label for="mytext1">Type a line of text, not more than 20 characters</label>
    <input type="text" name="mytext1" id="mytext1" required maxlength="20" minlength="10" value="" />
  </p>
  <p>
    <label for="mytext2">Type another line of text</label>
    <input type="text" name="mytext2" id="mytext2" value="This is content pre-inserted into the input field" />
  </p>
  <p>
    <input type="submit" name="submit" value="Submit" />
  </p>
</form>
```

This creates two single line inputs, one empty and the other pre-populated.

characters. The presence of the 'required' attribute means that the form cannot be submitted unless some text has been added, between 10 and 20 characters long.

The use of the label element 'for' attribute identifies the input element the label is attached to, which is particularly important for accessibility. For example, it would ensure that a screen reader could associate the label and entry field. It also helps usability as it allows you to click the label to select or enter text into the input element.

## Activity 2 Modifying the text input element

 Allow 15 minutes for this activity

From the snippets folder, open the file **text\_input.html** in your text editor. Change the values for `maxlength` and `minlength`, then save and load the file in the browser. Enter text that is less than your minimum and then greater than your maximum. Explore the response from the browser as you attempt to submit the form.

Notice that the second field is pre-populated with data but does not have a minimum or maximum set. You can add these if you wish.

Complete the form and submit your data to the reflector at <https://students.open.ac.uk/mct/tt284/reflect/reflect.php> from this form. Observe how the reflector identifies the data that you send.

## The textarea element

The next snippet we encounter is the text area input from the file **text\_area\_input.html**:

```
<form method="POST" action="https://students.open.ac.uk/mct/tt284/reflect/reflect.php">
  <p>
    <label for="mytextarea">Type a paragraph of text</label><br />
    <textarea name="mytextarea" id="mytextarea" rows="10" cols="30" required maxlength="100" minlength="10" />
  </p>
  <input type="submit" name="submit" value="Submit" />
</form>
```

```
</p>
<p>
  <label for="mytextarea2">Type another paragraph of text</label><br />
  <textarea name="mytextarea2" id="mytextarea2" rows="10" cols="30">
Text pre-inserted into a text area is added like this - inspect the code.</textarea>
</p>
<p>
  <input type="submit" name="submit" value="Submit" />
</p>
```

### Activity 3 Modifying the textarea element

 Allow 15 minutes for this activity

Edit this form to change the text in the label.

Note that if you change the 'ID' property of the element you must also change the label 'for' property to match.

Save your file, then load it in your browser.

Change the content of the second textarea element.

Observe that unlike text input elements which are self-closing and use a value attribute for pre-inserted text, textarea elements have an opening and closing tag with the value in between.

Observe how the reflector identifies the data that you send.

### The password input element

The password input works exactly like a text input but it hides the typed text. In this example, it has also been made a required element, with a minimum length of 12 characters, in the following snippet from your part 3 file **password\_input.html**.

```
<form method="POST" action="https://students.open.ac.uk/mct/tt284/reflect/reflect.php">
  <p>
    <label for="mypass">Type in your password, at least 12 characters</label>
    <input type="password" name="mypass" id="mypass" required minlength="12" />
  </p>
  <p>
    <input type="submit" name="submit" value="Submit" />
  </p>
</form>
```

## Activity 4 Modifying the password input element

 Allow 15 minutes for this activity

Edit the file **password\_input.html** to change the value of 'minlength'. Explore the response from the browser as you try and submit the form.

Test submission of the form without any input.

Now add a 'maxlength' attribute to the password element.

Save your file, then load it in your browser.

Observe how this changes what you see in the browser, and the information returned from the reflector when you submit the form.

Remove the 'required' attribute.


Save your file, then refresh your browser and test the submission of the form without any input.

## The number input element

Where the input value must be a number, then an input type number can be used. As well as restricting the values entered, the web browser may add buttons to increase or decrease the value. The relevant snippet inside the file **number\_input.html** is as follows (you'll note from the min and max values in the code that the form is looking for a number between 1 and 10 inclusive to be submitted):

```
<form method="POST" action="https://students.open.ac.uk/mct/tt284/reflect/reflect.php">
  <p>
    <label for="mycount">Select or type in a number:</label>
    <input type="number" name="mycount" id="mycount" min="1" max="10" />
  </p>
  <p>
    <input type="submit" name="submit" value="Submit" />
  </p>
</form>
```

## Activity 5 Modifying the number input element

 Allow 15 minutes for this activity

Edit the **number\_input.html** file to change the values of 'min' and 'max'.

Note that this form can be submitted without entering a value. Adding a 'required' attribute will force the HTML to validate the form at the client side. Edit the form and test the results in your browser.

Input text that is less than your minimum and then greater than your maximum. Explore the various responses from the browser as you try to submit the form.

## The select element

If we are to select just one item from a list of choices, then we can use a dropdown list, a radio button, or a button element. The select input element provides us with a dropdown list of options. Examining the relevant code in the part 3 file **select\_input.html**:

```
<form method="POST" action="https://students.open.ac.uk/mct/tt284/reflect/reflect.php">
  <p>
    <label for="flower">Select your favourite flower:</label>
    <select name="flower" id="flower">
      <option value="">Choose a flower</option>
      <option value="daisy">Daisy</option>
      <option value="tulip">Tulip</option>
      <option value="dahlia">Dahlia</option>
      <option value="rose">Rose</option>
    </select>
  </p>
  <p>
    <input type="submit" name="submit" value="Submit" />
  </p>
</form>
```

## Activity 6 Modifying the select input element

 Allow 15 minutes for this activity

Open the file **select\_input.html** in your editor.

Edit this file so that the value property isn't matched by the text that is displayed, for example, `<option value="daisy">Tulip</option>`

Save your file and then load it in your browser.

Observe the data that is returned by the reflector when you submit this form.

Note the importance of ensuring that the correct text is matched to each option.

## The radio input element

Another way of presenting a set of choices is to use "radio" buttons. Examine the file **radio\_input.html**. Note that in order to group inputs so that only one can be checked, each of them must have the same name attribute. The default choice is indicated by the 'checked' attribute. We have used a fieldset element to present

the radio buttons in a group and have given this a legend, as noted earlier, which is useful for accessibility purposes. Again, note the importance of checking that the displayed text matches the value being sent to the server when the form is submitted.

```
<form method="POST" action="https://students.open.ac.uk/mct/tt284/reflect/reflect.php">
  <fieldset>
    <legend>Select your favourite colour:</legend>
    <input type="radio" name="colour" id="red" value="red" checked />
    <label for="red">Red</label><br />
    <input type="radio" name="colour" id="blue" value="blue" />
    <label for="blue">Blue</label><br />
    <input type="radio" name="colour" id="green" value="green" />
    <label for="green">Green</label><br />
  </fieldset>
  <p>
    <input type="submit" name="submit" value="Submit" />
  </p>
</form>
```

## Activity 7 Modifying the radio input element

 Allow 15 minutes for this activity

Open the **radio\_input.html** in your editor.

Change the 'name' of just one of these radio elements. Save your file, load it in your browser and observe what happens as you select each radio element, then submit the form.


## The checkbox input element

All the choices in a checkbox input also have the same name, but this time any number from zero to all of them can be selected. Again a default choice can be indicated by the 'checked' attribute, in this case in the file **checkbox\_input.html** we've coded apple as a pre-checked input, and we have used a fieldset element to present the checkboxes in a group and given this a legend. The pair of square braces for the name attribute are required if they are to be processed by PHP on the server side. Code to process this is provided later in the block.

```
<form method="POST" action="https://students.open.ac.uk/mct/tt284/reflect/reflect.php">
  <fieldset>
    <legend>Tick all the fruit you like:</legend>
    <input type="checkbox" name="fruit[]" id="apple" value="apple" checked />
    <label for="apple"> Apple</label><br />
    <input type="checkbox" name="fruit[]" id="banana" value="banana" />
    <label for="banana"> Banana</label><br />
    <input type="checkbox" name="fruit[]" id="grape" value="grape" />
    <label for="grape"> Grape</label><br />
  </fieldset>
  <p>
```

```
<input type="submit" name="submit" value="Submit" />
</p>
</form>
```

## Activity 8 Modifying the checkbox input element

 Allow 15 minutes for this activity

Edit the file **checkbox\_input.html** to add two more checkbox elements that use `name="fruit[]"` , then add another two checkbox elements that use a different value for the 'name' property.

Save your file, then refresh your browser.

Observe what happens to the data when you submit the form each time. In particular, notice when the name is the same the responses are combined in a single array.

## The select multiple element

To select more than one item, where there are more choices than can be provided using checkboxes, a dropdown list that allows multiple selection may be used. This done by adding the 'multiple' attribute to the select input we already saw, as demonstrated in the **select\_multiple\_input.html** file. Hold down the Ctrl (Windows) / Command (Mac) button to select multiple options.

```
<form method="POST" action="https://students.open.ac.uk/mct/tt284/reflect/reflect.php">
  <p>
    <label for="mflower">Select all the flowers you like:</label>
    <select name="mflower[]" id="mflower" multiple size="4">
      <option value="daisy">Daisy</option>
      <option value="tulip">Tulip</option>
      <option value="dahlia">Dahlia</option>
      <option value="rose">Rose</option>
    </select>
  </p>
  <p>
    <input type="submit" name="submit" value="Submit" />
  </p>
</form>
```

## The hidden and submit input elements

Sometimes we might want a form to include data that we have added at the time the form was created, or added or changed by JavaScript, but it isn't necessary for the user to see or edit this data; it might be information about a discount the user is entitled to or other data that might confuse the user. In such cases, we implement the hidden form element.



We may also wish for the user to submit the same form but to carry out a different task using the submitted data. We could offer this choice using radio buttons and a single submit button, or we can vary the 'value' attribute of the submit button. Unlike hidden inputs, this value is shown to the user.

These are both demonstrated in the file **hidden\_input.html**:

```
<form method="POST" action="https://students.open.ac.uk/mct/tt284/reflect/reflect.php">
  <input
    type="hidden"
    name="myhidden"
    id="myhidden"
    value="This content is hidden from the user - but note they can read and change it in the page source
  />
  <p>
    <input type="submit" name="task" value="save" />
    <input type="submit" name="task" value="edit" />
    <input type="submit" name="task" value="delete" />
  </p>
</form>
```



## Activity 9 Examining the hidden form input

 Allow 15 minutes for this activity

Open **hidden\_input.html** in your browser.

Add a second hidden element to this form. It will require a name and ID that is not the same as any existing element in the form. Add your own content to your hidden field.

Save your file, then load it in your browser.

Now submit this form and observe what is returned from the reflector. Try each of the submit buttons in turn.

From this you should be able to identify the content sent from the hidden and submit fields.


## The email input element

Entering an email address is a common requirement. These are text so can be entered using a text input, but it is easy to enter an invalid email address. An email input provides a simple way to request that the web browser validates an email address without having to create your own pattern-matching code.

```
<form method="POST" action="https://students.open.ac.uk/mct/tt284/reflect/reflect.php">
  <p>
    <label for="myemail">Email:</label>
    <input type="email" name="myemail" id="myemail" value="" />
  </p>
```

```
<p>
  <input type="submit" name="submit" value="Submit" />
</p>
</form>
```

## Activity 10 Examining the email form input

 Allow 10 minutes for this activity

Client-side validation of email is unsophisticated in that any content which contains a single '@' symbol, but does not start or end with one, or contain more than one, is likely to be accepted.

Test this in your browser using the file **email\_input.html**. Can you find other email address strings that are accepted or rejected on the client side? Report any findings you might have to the Block 2 Forum.

Email validation is difficult as the rules on what constitutes a valid email address are surprisingly complicated. The best way of validating an email address is to send a message to that address and ask the recipient to confirm that they have received it. However, that is beyond the scope of the module.

## Summary

You have now met and tried a range of HTML input types and experienced their strengths and weaknesses. This list is not exhaustive – there are other input types available that the web browser may assist the user in entering, and validate the value entered, particularly on mobile devices. The HTML standard continues to evolve, and new input types and attributes are added as it does so. These input types can often be sufficient for your client-side validation.

## 3.2 Adding styles to a form

### Activity 11 Examining styles

 Allow 5 minutes for this activity

For this next section, you need to open two files from the Block 2 Part 3 resources in your text editor at the same time, **html-forms-styled.html** and **html-forms.css**.

Look inside the head element of the HTML file and you will see the link element that identifies our CSS file to be used:

```
<link rel="stylesheet" href="html-form.css">
```

This is the main change from the html-forms.html file we started with, but we have also added two class attributes which you will need later.

The default form elements are quite utilitarian, but we should always design our web page or user interface to be usable without the use of CSS.

We can, however, enhance the appearance and sometimes the usability of form elements by making use of CSS.

We could give all input elements the same style by including the following in our style sheet:

```
input {  
    background-color: #F2F6D7;  
    width: 100%;  
}
```

Recall that the colour can be given as a hexadecimal value representing the proportions of red, green and blue. The first two digits represent the red (F2 hex is 242 decimal) the next two green (F6 = 246 decimal) and the final two blue (D7 = 215 decimal).

When such a style rule is added at the top of a style sheet, that style will prevail through all form input elements, except in cases where other styles are specified later for certain form elements; these additional styles are placed lower down in the hierarchy of the style sheet. That cascade of style from top to bottom is what gives cascading style sheets their name.

CSS can be made more specific by using selectors that look for attributes such as the input type, followed by additional rules to apply to elements of that type, for example:

```
body {  
    background-color: #d7f2f6;  
}
```


```
input {  
    background-color: #f2f6d7;  
    width: 200px;  
}
```

```
input[type=submit] {  
    width: 100px;  
    height: 2em;  
    background-color: #96d09e;  
}
```

```
input[type=text] {  
    background-color: #ffffff;  
    height: 20px;  
    padding: 15px 25px;  
    margin: 6px 0;
```

```
}  
input[type=password] {  
    background-color: #f6d7f2;  
}
```

## Activity 12 Modifying CSS styles

 Allow 15 minutes for this activity

Edit the CSS file that you have already opened in your text editor: **html-forms.css**

The colours are represented as hexadecimal hash versions. Change the colour in each of the input element styles above; try using RGB versions or named colour versions to see how the CSS affects the design of the HTML form. Make your changes, then save the CSS file to reflect them. You will need to refresh the HTML page in your browser before it displays any changes as static HTML pages are rendered as they load and do not refer back to the source files to look for changes. That means the page can remain displayed even if the connection it loads.

Be sure you can reliably style a chosen element.

Some input elements are much easier to style if you add a class to the form element.

We have added a class 'radio\_set' and another 'checkbox\_set' to the fieldset elements enclosing the radio buttons and check boxes.

In our CSS we can create a style for those classes

```
.radio_set{  
    background-color: #00ffe2;  
}
```

```
.checkbox_set{  
    background-color: #ff00e2;  
}
```

## 4 Client-side validation

We have seen how HTML elements and attributes can help control data entry – preventing invalid values being entered and submitted to a web server. While HTML can solve simple client-side validation issues, there are still instances where JavaScript can assist. HTML delivers the content, CSS presents the content and JavaScript manipulates the content. In response to input events, scripts will execute and manipulate select elements of the HTML or properties of the CSS to provide an interactive experience. Scripting on the client-side places less stress on the server as the processing work is done on the client's processor.

## What if JavaScript is not available?

If you include the following in the HTML code, the browser will display the warning message if JavaScript is not enabled:

```
<noscript>
```

```
JavaScript is not enabled.
```

```
Some features of this page will not be available.
```

```
</noscript>
```

We have provided a **noscript.html** file and if you wish you can test this by disabling JavaScript using the developer tools we described in Block 1.

Press F12 to open the tools, choose the Sources tab. Now press ctrl+shift+p to run a command. Start to type JavaScript and choose the orange button to disable JavaScript as shown in Figure 1.

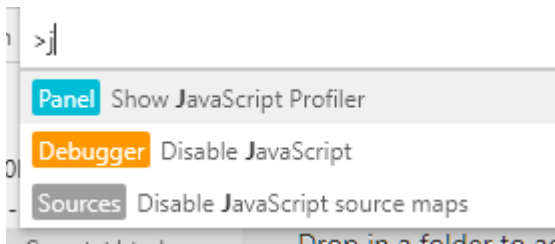


Figure 1 Disable JavaScript

### Activity 13 Is user input on the client side worth correcting?

 Allow 15 minutes for this activity

There are several benefits to be gained from using JavaScript on the client side to be able to validate the data that have been entered in the fields of a form.

Make a brief list of at least three benefits that occur to you.

Then consider the following:

- If the form data is validated on the client side, does this mean that when data reaches the server, there is no need for the server to validate the data again?

Discuss your reasoning in the Block 2 Forum.

Reveal answer

## 4.1 Validation step by step

The validation process iterates through a series of distinct steps summarised below:

1. Detect that the user has changed an input value through typing or clicking by responding to form events.
2. In JavaScript, obtain a reference to the input element and its value.
3. Compare that value against what is permitted for that input element and the business requirements of the application.
4. If the value matches the requirements, provide feedback to the user to confirm this.
5. If the value doesn't match the requirements, provide a different message, indicating that the data is not suitable, and explaining what is needed.
6. If the user attempts to submit the form to the web server while some input values don't match the requirements, the submission is blocked and a further message explaining what needs to be done is added.
7. If the user attempts to submit the form when all input values match the requirements, then the form submission is permitted.

Over the next few sections, we will explore JavaScript code that goes through each of these steps.

## 5 Responding to form events

Events are triggered by user actions within form elements on a web page; we can illustrate this to some extent by engaging with the file `form-events.html`.

### Activity 14 Examining JavaScript events

 Allow 30 minutes for this activity

Open the file **form-events.html** in your browser.

Then follow the instructions in the page to click or add some content to the form input elements. The instructions also state which event is being detected. Open this file in your chosen text editor and examine the HTML code and the JavaScript code as you read through the following explanations.

### Input events

Inside the form element you should see that each input element has either an 'onfocus', 'onblur', 'onchange' or 'oninput' attribute with a value containing some code to call a JavaScript function. These 'on' attributes correspond to JavaScript events of the same name: 'focus', 'blur', 'change' and 'input'.

For example, to handle a 'focus' event on an input element, we add an 'onfocus' attribute:

```

<p>
  <label for="testfocus">Enter some text:</label>
  <input type="text" id="testfocus" onfocus="handleEvent(event)" />
  <span id="feedback_testfocus"></span>
  then click somewhere else on the page
</p>

```

In the code above, each time a 'focus' event occurs for the input element, a function named 'handleEvent' will be called. It is passed a single variable, named 'event'. This variable is a JavaScript object with a set of properties describing the event which occurred. This includes a reference to the input element for which the event occurred, known as the 'target' of the event.

These events are often used to detect that data has been added entered or changed:

- **focus**: which occurs when an element gets focus, for example, the user uses the Tab key to 'tab' to an input or clicks on a text input or its label.
- **blur**: which occurs when an element loses focus, for example, the user clicks somewhere else on the page or field.
- **change**: which occurs when the value of an element changes at the point the change is completed, for example, by clicking or by losing focus after entering text. This event is particularly useful for detecting that the user has made a new selection from a dropdown list, set of radio buttons or checkboxes.
- **input**: which occurs when the value of an element changes, each time it happens, for example, on every keypress. This event may not be supported in all web browsers, and the user may not welcome feedback before they have finished entering data, so we will use 'change' instead.

The 'handleEvent' function in **form-events.html** show how the event and the properties of its target such as 'id' and 'value' are received by the function and can then be used in further actions. Passing events or event targets to a function in this way means that we don't need a separate function for every form field:

```

function handleEvent(event) {
  var target = event.target;
  var id = target.id;
  var value = target.value;
  console.log("sent type=" + event.type + " for id=" + id + " value=" + value);
  var feedbackId = "feedback_" + id;
  document.getElementById(feedbackId).innerText = event.type + " detected!";
}

```

Observe the following key points:

- The 'event' object has a 'type' property with the name of the event that occurred, for example, 'change'.
- The 'event' object has a 'target' property which is the input element that the event occurred for. Like 'event', this is also a JavaScript object.
- The input element has 'id' and 'value' properties which can then be read to discover what the current value of the input is, and what it should be, for example, by adding code that is executed if the id is equal

to 'firstname'.

- Feedback is inserted after the input element by changing the text content of the span with an id attribute corresponding to the input id attribute, for example, 'feedback\_firstname'.

Note we can also show that an element has focus using CSS. We have set the files with focus yellow, and also used CSS to set the background colour of any span element immediately following an input element, so that our feedback messages are easier to see:

```
input:focus {  
    background-color: yellow;  
}  
input + span {  
    background-color: greenyellow;  
}
```

Reload the file in your web browser and press the Tab key on your keyboard to move through the inputs. Try entering different values and observe again when each event is triggered, the message output to the JavaScript console, and the feedback text inserted into the page.

## Form submission

Next, we will consider another useful event, but this time it is an event that occurs for form elements not input elements. This is the `submit` event which occurs when a form is submitted. This is very useful to trigger a check of the completed form before it is submitted and prevent submission if necessary. Note that `onsubmit` is an attribute of the form element and not the submit button.

This JavaScript code is slightly different to our earlier example:

```
<form  
    method="POST"  
    action="https://students.open.ac.uk/mct/tt284/reflect/reflect.php"  
    onsubmit="return handleSubmit(event.target)"  
>
```

Observe these three differences:

- The function is named 'handleSubmit' and is intended to handle only this specific event.
- Instead of passing the 'event' object, we pass its 'target' property. As we know which event to expect we are only interested in which element the event occurred for, that is, which form element was submitted.
- The function call is prefixed with the 'return' keyword. This allows the event handler to change the web browser behaviour – if the function returns a false value, the web browser will cancel the form submission.

This script is also demonstrating a technique to detect and iterate through all the text input elements in the form using the 'querySelectorAll' method we encountered in Part 2. This is quite advanced so don't worry too much about the exact syntax but focus on the information the script adds to the page when you try to submit



the form.

The key points to take from this are that we can potentially use JavaScript both to provide feedback as individual form inputs are used, and to intercept a form submission, make the same checks, and cancel form submission if data is invalid.

The last line always returns false and so our form will never be submitted. We will change that later, but for now it allows us to see the results we have written to the webpage.

## Attaching event handlers

There are three main methods of triggering JavaScript when document events occur, listed here in order of when they were invented.

1. Originally, within the HTML code itself. Although this method is no longer recommended, it is the simplest method, and we will continue to use it in some examples. In this example, when a 'change' event occurs, our 'onchange' code will call the named 'handleEvent' function passing an 'event' object as the first argument.

```
<input type="text" id="testchange" onchange="handleEvent(event)" />
```

2. More recently, by using the DOM properties to assign an event handler to an HTML element.

Here the form input element has an 'id' attribute but there is no JavaScript in the HTML code. Instead, the event is attached using a separate JavaScript instruction. This allows the developer to change the event by editing the JavaScript code, which may be in a JavaScript file, rather than editing the HTML. Unlike the first method, however, you cannot tell which events are handled simply by examining the HTML code. In this example, when a 'change' event occurs, JavaScript will call the named 'handleEvent' function passing an 'event' object as the first argument. Note that the 'handleEvent' function is referred to by name, without round brackets that would call the function and use its return value. .

```
<input type="text" id="testchange" />
```

```
<script>
document.getElementById("testchange").onchange = handleEvent;
</script>
```

3. Most recently, by using the 'addEventListener' method to attach an event handler to an HTML element. Again, you cannot tell simply by looking at the HTML source code if an event is handled, but you can add more than one event handler for a single event, for example, to add extra behaviour to an existing page. As with the previous example, when a 'change' event occurs, JavaScript will call the named 'handleEvent' function passing an 'event' object as the first argument.

```
<input type="text" id="testchange" />
```

```
<script>
```

```
document.getElementById("testchange").addEventListener("change", handleEvent);
</script>
```

Note that the implementation of the 'handleEvent' function is the same in each of the above examples: it is always passed an 'event' object. When connecting an event handler using an HTML 'on' attribute this is obtained from the global JavaScript 'event' variable. When using the two more recent methods, JavaScript itself passes the 'event' object.

## 6 Collecting the data item for analysis

You can see that each of our functions that detects the user has made some sort of input then passes a reference to the element and the value it contains to our checking function in our JavaScript.

This avoids the need to write a separate function for each input field. We can also place our validation functions in a library that we can use for many different forms.

Although it might seem to be the simplest to write a separate function to check each input field, over the life of a website this can result in a lot of extra maintenance work and development time.

When we design our set of forms for a website or a web page, it is necessary to consider what defines a valid input and separates it from an invalid input. We also need to plan what extra feedback might be needed should the input not be valid.

For example, **input plan for a new club membership form:**

	id and name  (no hyphens)	Input type	HTML attributes	What makes a valid entry?	Feedback	Notes
All members must agree that they will abide by the rules of the club and accept the conditions of membership and pay all fees due	agreement	checkbox	HTML required	ticked		Feedback message provided by web browser

First name	firstname	text	HTML minlength=2  maxlength=20	A minimum of two characters and a maximum of 20 characters	First name is required. At least two characters and no more than 20 characters	
Last name	lastname	text	HTML minlength=2	A minimum of two characters	Last name is required. At least two characters	
Address house name or number	house	text	HTML minlength=1	A minimum of one characters	Name or number of house, required	
Address 2	add2	text	HTML minlength=2	A minimum of two characters	Address is required	
Address 3	add3	text	None			This field is optional
Town	town	text	HTML minlength=2	A minimum of two characters	A town is required	
Postcode	postcode	text	None, use JavaScript to validate	Matching any valid postcode	A valid postcode is required	
UK Telephone landline	tellandline	text	None, use JavaScript to validate.  We could also use a HTML 'tel' input	Min 10, max 11 digits  starts with 0  strip spaces and brackets	Please enter as just the number without any 44 prefix or spaces	Note we are not covering all the possible ways that users can type a telephone number!

UK Telephone mobile	telmobile	text	None, use JavaScript to validate.	Min 10, max 11 digits	Please enter as just the number without any spaces
			We could also use a HTML 'tel' input		
email	email	text	None, use JavaScript to validate.	A minimum of 6 characters	name@domain
			We could also use a HTML 'email' input		

---

## 7 Providing feedback

Feedback is an essential part of the design of any web form. In Block 2 Part 2, we looked at setting the 'innerText' property of a specific element to insert content into the page. This is one of the most efficient ways of giving feedback to the user. When we design a web form, we also design the elements where each item of feedback will go. It is also very useful to use a standard naming convention for the ID of the feedback element that allows it to be related to the ID of the corresponding form input element, perhaps by adding 'feedback\_' to the front. An ID of 'firstname' in the form becomes an ID of 'feedback\_firstname' in the span element for the reply.

Building on our previous example, each input now looks like:

```
<p>

<label for="firstname">First name:</label>
<input
  type="text"
  id="firstname"
  name="firstname"
  onchange="validate(event.target)"
/>
<span id="feedback_firstname" class="invalid"></span>
</p>
```

It is also very useful to check a form entry and give feedback to the user immediately after they have entered a value. That is, as soon as the user moves away from that element – that element loses focus. Provide a positive response if it is a valid input – such as a green tick alongside the entry, or a comment on the strength of a password. This helps the user spot which form elements still need an input.

Incorrect values need a clear display that indicates that they need to be changed, as well as a message that indicates how they should be changed or why the entered value is not suitable.

There are other feedback methods, but they are less suited to data entry: `console.log` is very useful when we are debugging a script. Alerts can be useful, but do not look very professional on a finished web page and do not provide good usability.

It is possible to dynamically create the new HTML elements for supplying feedback, but this is not a good idea. It creates problems for screenreaders and so reduces accessibility as well as making it more complex to update each time the user adds more data to the form as you need to remember to remove the last element you added.

## 8 Checking that a data item meets the criteria

In JavaScript, we need a clear convention for our validation functions, otherwise we can end up with errors in our code that are tricky to track down. This convention should also be clear to anyone else looking at the code.

We are asking if our data is valid, our convention will be that each function should return `true` if it is valid and `false` if it is not valid.

JavaScript allows us to check for things such as whether data is numeric or textual (a string value), the range of numeric data, the contents of string data, or whether any data has been entered at all. We can use 'regular expressions' to verify whether a value meets more complex requirements (e.g. a registration number for a vehicle).

### Testing a user has entered some data.

A user may have clicked on an input field, but then left it empty. They may also have added space characters, but still no useful data. We need to trim the data item to remove any leading or trailing spaces before we check.

There are two possible values for an empty field; a null value or a zero length string. We need to check for them both. If the input field can hold any string of characters then this is all the checking we need to do. The function returns `true` if the data item is not null and is not empty.

We have already seen that this can be done using the 'required' or 'minlength' attribute of a text input, but we also need to know how to check this using JavaScript for situations where, for example, only certain characters are allowed or not allowed.

### 8.1 Pattern matching using regular expressions

Pattern matching can be done using regular expressions, otherwise known as 'regex'. The regex is a special string of characters that creates a search pattern. A 'test' or 'match' of the regex compares this search pattern with the text string being tested. The simpler regex search patterns we can create ourselves, but for more complex patterns, like a regex for valid post codes, it is better to look them up, or in real applications, use a database of valid postcodes.

Each regex consists of a `'/'` character with a pattern between the two. It may also be followed by 'flag' characters that change how the match works.

At its simplest, we can look for an identical string of characters to the regular expression.

In this first example, we are looking for a match with the word 'fox'.

In JavaScript, we create a regex object as follows:


```
var regex = /fox/i;
```

and test it against the value of an input element with the line:

```
var result = regex.test(inputElement.value);
```

The result will be true if the value matches the pattern, or false if not. Note that the `/i` following the pattern indicates to ignore case when matching. While the comparison is simple, we already have gained power by being able to match without worrying about upper- or lower-case letters.

## Activity 15 Use JavaScript to validate user input

 Allow 15 minutes for this activity

Open the file **pattern-match.html** in your text editor and in your browser. The first text input contains the word 'fox', but when the page is loaded the JavaScript is not called because the input box has not changed. Change the value but leave the word 'fox' in place and observe the feedback that appears.

Now change the word 'fox' to something else. Notice that the feedback updates as you type – this is because JavaScript is listening for the 'input' event (also the 'change' event in case the 'input' event does not work).

Now try the second text box, which is set to look **only** for the word 'fox'. Observe that there is no match until you remove all the text except the word 'fox'.

Open the file in your editor and edit the lines that set the regular expression patterns to match and change 'fox' to something else.

Save your file and reload in your browser.

Test the input with a sentence that contains your word, then with a sentence that doesn't contain your word, and then with only your word.

The difference in behaviour is due to 'anchors'. If we want to check that a value contains text matching a pattern, we do not need anchors, but if we want to check that the whole of a value matches a pattern, or that the pattern matches the whole of the value, as is the case when validating form input, we need to add anchors to the pattern.

The characters ^ and \$ are used as anchors, and they are important to avoid unexpected results. An anchor ties a regular expression to the start (or end) of the value. If we want to test if a telephone number begins with '09' indicating a premium rate, just testing /09/ will match '09123456789' as we expect, but also match '01908123092' which we don't want. It matches the second telephone number because it contains the digits '09' near the end.

To avoid that we can add a ^ character to anchor our test to the start of the value, so the expression /^09/ will only match strings with the digits '09' at the start. Adding a \$ character will anchor our test to the end of the text, so /09\$/ will only match strings ending in '09'.

Returning to the 'fox' example, to match a value that contains the word 'fox' then /fox/ is enough, but to ensure the value contains the word 'fox' and nothing else, then the pattern is /^fox\$/.

This pattern only matches a single word – we will move on to more complex examples, but these must always have anchors to provide useful input validation.

## Reusability through consistency

The JavaScript code in this example is starting to become reusable. Each text input calls the same JavaScript function when it changes, and the code checks the 'id' attribute of the input element to decide what pattern to match. The input is prefilled using the 'value' attribute. For example:

```
<p>
  <label for="testfox">Look for a fox:</label>
  <input
    type="text"
    id="testfox"
    value="The quick brown fox jumps over the lazy dog"
    onchange="validate(event.target)"
    oninput="validate(event.target)"
  />
  <span id="feedback_testfox"></span>
</p>
```

It makes it much easier to keep track of each component and ensure that they match each other if we specify our naming system. In this example, the 'id' of the input is used to decide which pattern to match:

```
var pattern;
if (id == "testfox") {
  pattern = /fox/i;
}
```

The same code can then check either input:

```

if (pattern.test(value)) {
    document.getElementById(feedbackId).innerText =
        "Found a match for " + pattern.source;
} else {
    document.getElementById(feedbackId).innerText =
        "No match for " + pattern.source;
}

```

If we look through the code we can find the locations where 'testfox' is used, for example:


- `<label for="testfox">`
- `<span id="feedback_testfox">`
- `<input type="text" id="testfox">`

This code will become more useful if we can vary the feedback message according to the input, which we will do next.

## 9 A working example of form validation

The following activity implements the membership form we discussed in an earlier section, building on the pattern matching example in the previous section to add different feedback as well as a different pattern for each input element. It uses CSS classes to indicate valid and invalid feedback. Finally, it validates all the inputs again when form submission is attempted.

### Activity 16 Exploring the membership form example

 Allow 30 minutes for this activity

Open the file **membership-form.html** in your text editor and in your browser.

Also open the file **membership-form.js** and **membership-form.css** in your text editor.

Both files are in the download from the Block 2 Part 3 Resources.

Now explore what happens when you enter values into the form inputs.

**Then examine the code to see how that form input is being validated as it is entered and when the form is submitted, and how feedback is provided to the user.**

When reading the JavaScript, look for the following features:



- Each input is validated by calling a 'validate' function passing the target of the change event.
- The 'validate' function then examines the 'id' and 'value' properties of that element to decide what regex to use and what feedback to provide if the value is invalid.
- The feedback is provided by getting a reference to the DOM element with an id corresponding to the input element and setting its 'innerText' property.
- The feedback colouring is implemented using two CSS classes 'valid' and 'invalid' and setting the 'className' property of the feedback element – which corresponds to the 'class' attribute you can set in HTML.
- When the form is submitted, a 'validateForm' function is used. Unlike earlier examples this function has no parameters as it is written specifically for the membership form.
- The 'validateForm' function calls the 'validate' function for each input element, as if each one received a 'change' event in turn. Only if all of these function calls return true, will the 'validateForm' function return true, and the web browser be permitted to submit the form.

When using the form, observe the interaction between HTML and JavaScript validation:

- HTML validation messages only appear when the form is submitted, while JavaScript validation messages can appear at any time in response to 'change' (or 'input' or 'blur' events – you can add these if you wish), helping the user fix errors as soon as they make them rather than waiting until they submit the form.
- HTML validation messages appear only for the first input that is invalid while JavaScript can add feedback after all the input elements, making it easier for the user to see what is wrong, and not have to repeatedly try to submit the form.

## 10 Regex reference

Regular expressions are used in very similar ways by many different programming languages, but beware: they are not always used in an identical way. The easiest way to test regular expressions is using a free online tool – search the web for 'regex tester' to discover these.

A regular expression is a string of characters that represents a pattern that should be searched for in another string – the target string. Remember that for input validation it is important to use ^ and \$ anchors to ensure you match inputs that contain *only* the value you expect.

---

<b>/abcd123/</b>	Will only match abcd123. Some characters just match the same character in the target string.
<b>/ab.d1/</b>	The dot here matches any characters, so it would match e.g. ab£d1 or abcd1 or ab6d1 and so on.
<b>/ab\dab/</b>	\d represents any single digit, so it would match e.g. ab4ab or ab7ab, but not abcab.
<b>/ab\wab/</b>	\w represents any alphanumeric character e.g. abDab or abcab or ab3ab, but not ab?ab.
<b>/\d\d\d\d/</b>	Represents a string of four consecutive digits anywhere in the target string e.g. aaaaaaaaa1234bbbbbbbbbb.
<b>/^\d\d/</b>	The ^ character anchors the search to the start of the target string.

<code>\d\d\$</code>	The \$ character anchors the search to the end of the target string. It will match any string that ends with two digits.
<code>^\d\d\$</code>	This expression is anchored to both start and end. It will only match a string of exactly two digits.
<code>/a+/</code>	The + matches one or more of the preceding character. This will match a single a character or a long string of aaaaa characters.
<code>/a*b/</code>	The * matches zero or more of the preceding character. So <code>/a*b/</code> matches 'b' as well as 'ab' and 'aaaab'.
<code>/the.*</code>	These matching characters can be joined together, so this will match the string 'the' followed by zero or more of any characters.
<code>/n{3}/</code>	{3} indicates repetition of the previous character. This will match 'nnn', but not 'nn'.
<code>\d{10}</code>	Again these can be combined. This will match a string of ten digits.
<code>\d{5,10}</code>	This will match a string of five to ten digits.
<code>/[ace]/</code>	Square brackets denote a 'character class'. Matches any single character from within the square brackets.
<code>/[0-9]/</code>	Matches any single digit.
<code>/[a-z]/</code>	Matches any single lower case character from a to z.
<code>/[a-zA-Z]/</code>	Matches any single letter in lower or upper case.
<code>/[a-zA-Z]{2,6}/</code>	Matches two to six letters (they need not be the same letters).
<code>/(fox cat)/</code>	Round brackets denote alternative patterns, separated by a 'pipe' (vertical bar) character. Matches 'fox' or 'cat'.
<code>/(fox cat dog)/</code>	Matches 'fox' or 'cat' or 'dog'.
<code>\\\$\\^</code>	If we want to match any of the characters that have special meaning in the regex expression we have to escape them. That means using the \ character in front of them. Therefore this should match a string that looks like <code>\\$^</code> .

#### Other useful matches are:

<code>\s</code>	Whitespace characters.
<code>\S</code>	Not a whitespace character.
<code>\D</code>	Not a digit.
<code>\t</code>	Tab character.
<code>\r</code>	Carriage return.

`\n` New line character. Note Windows often uses `\r\n` in text files for a new line.

---

**Regex expressions can also use a number of flags. e.g.**

---

<code>/a*b/g</code>	The <code>g</code> here indicates a global match, not stopping at the first occurrence.
<code>/a*b/i</code>	The <code>i</code> here indicates ignore case, so this will match A and B or A and b etc.
<code>/a*b/m</code>	The <code>m</code> here indicates a search over multiple lines.
<code>/a*b/gi</code>	More than one flag can be used at a time.

---

Regular expressions have their uses, but they are not always the right solution. They can become long and difficult to understand. For some real-world data such as postcodes and email addresses, it is often more useful to validate by looking up in a postcode database or by sending an email and confirming that it was received, but that is beyond the scope of this module.

## 11 Summary

In this part, we focused on using forms as the main user interface for entering data into a web application. We have worked through the steps taken to perform client-side validation and feedback using JavaScript.

## References

Mozilla (2023) *Regular Expressions*. Available at: [https://developer.mozilla.org/en/docs/Web/JavaScript/Guide/Regular\\_Expressions](https://developer.mozilla.org/en/docs/Web/JavaScript/Guide/Regular_Expressions) (Accessed: 3 June 2023).

W3C (2023) *HTML5 A vocabulary and associated APIs for HTML and XHTML* W3C Recommendation. Available at: <https://www.w3.org/TR/html5/forms.html#forms> (Accessed: 3 June 2023).

Mozilla (2023) *HTML5 form updates*. Available at: <https://developer.mozilla.org/en-US/docs/Learn/HTML/Forms> (Accessed: 3 June 2023).

## Where next?

We now have a good knowledge of how to handle web forms and provide user feedback on the client side.

In Block 2 Part 4 we will see how to generate the forms and handle the data on the server side.

---